# 29｜接口：为什么nil接口不等于nil？

2021-12-29 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

**讲述：Tony Bai**
时长 22:29 大小 20.59M

你好，我是 Tony Bai。

上一讲我们学习了 Go 接口的基础知识与设计惯例，知道 Go 接口是构建 Go 应用骨架的重要元素。从语言设计角度来看，Go 语言的接口（interface）和并发（concurrency）原语是我最喜欢的两类 Go 语言语法元素。Go 语言核心团队的技术负责人 Russ Cox 也曾说过这样一句话：“**如果要从 Go 语言中挑选出一个特性放入其他语言，我会选择接口**”。这句话足以说明接口这一语法特性在这位 Go 语言大神心目中的地位。

为什么接口在 Go 中有这么高的地位呢？这是因为**接口是 Go 这门静态语言中唯一“动静兼备”的语法特性**。而且，接口“动静兼备”的特性给 Go 带来了强大的表达能力，但同时也给 Go 语言初学者带来了不少困惑。要想真正解决这些困惑，我们必须深入到 Go 运

领资料

行时层面，看看 Go 语言在运行时是如何表示接口类型的。在这一讲中，我就带着你一起深入到接口类型的运行时表示层面看看。

好，在解惑之前，我们先来看看接口的静态与动态特性，看看“动静皆备”到底是什么意思。

## 接口的静态特性与动态特性

接口的**静态特性**体现在**接口类型变量具有静态类型**，比如`var err error`中变量 err 的静态类型为 error。拥有静态类型，那就意味着编译器会在编译阶段对所有接口类型变量的赋值操作进行类型检查，编译器会检查右值的类型是否实现了该接口方法集合中的所有方法。如果不满足，就会报错：

复制代码

```
1 var err error = 1 // cannot use 1 (type int) as type error in assignment: int
```

而接口的**动态特性**，就体现在接口类型变量在运行时还存储了右值的真实类型信息，这个右值的真实类型被称为接口类型变量的**动态类型**。你看一下下面示例代码：

复制代码

```
1 var err error
2 err = errors.New("error1")
3 fmt.Printf("%T\n", err)  // *errors.errorString
```

我们可以看到，这个示例通过 errros.New 构造了一个错误值，赋值给了 error 接口类型变量 err，并通过 fmt.Printf 函数输出接口类型变量 err 的动态类型为 *errors.errorString。

那接口的这种“动静皆备”的特性，又带来了什么好处呢？

首先，接口类型变量在程序运行时可以被赋值为不同的动态类型变量，每次赋值后，接口类型变量中存储的动态类型信息都会发生变化，这让 Go 语言可以像动态语言（比如 Python）那样拥有使用 Duck Typing（鸭子类型）的灵活性。所谓鸭子类型，就是指某类型所表现出的特性（比如是否可以作为某接口类型的右值），不是由其基因（比如 C++ 中的父类）决定的，而是由类型所表现出来的行为（比如类型拥有的方法）决定的。

比如下面的例子：

复制代码

```
type QuackableAnimal interface {
    Quack()
}

type Duck struct{}

func (Duck) Quack() {
    println("duck quack!")
}

type Dog struct{}

func (Dog) Quack() {
    println("dog quack!")
}

type Bird struct{}

func (Bird) Quack() {
    println("bird quack!")
}

func AnimalQuackInForest(a QuackableAnimal) {
    a.Quack()
}

func main() {
    animals := []QuackableAnimal{new(Duck), new(Dog), new(Bird)}
    for _, animal := range animals {
        AnimalQuackInForest(animal)
    }
}
```

这个例子中，我们用接口类型 QuackableAnimal 来代表具有“会叫”这一特征的动物，而 Duck、Bird 和 Dog 类型各自都具有这样的特征，于是我们可以将这三个类型的变量赋值给 QuackableAnimal 接口类型变量 a。每次赋值，变量 a 中存储的动态类型信息都不同，Quack 方法的执行结果将根据变量 a 中存储的动态类型信息而定。

这里的 Duck、Bird、Dog 都是“鸭子类型”，但它们之间并没有什么联系，之所以能作为右值赋值给 QuackableAnimal 类型变量，只是因为他们表现出了 QuackableAnimal 所要求的特征罢了。

不过，与动态语言不同的是，Go 接口还可以保证“动态特性”使用时的安全性。比如，编译器在编译期就可以捕捉到将 int 类型变量传给 QuackableAnimal 接口类型变量这样的明显错误，决不会让这样的错误遗漏到运行时才被发现。

接口类型的动静特性让我们看到了接口类型的强大，但在日常使用过程中，很多人都会产生各种困惑，其中最经典的一个困惑莫过于“nil 的 error 值不等于 nil”了。下面我们来详细看一下。

## nil error 值 != nil

这里我们直接来看一段改编自GO FAQ 中的例子的代码：

复制代码

```
type MyError struct {
    error
}

var ErrBad = MyError{
    error: errors.New("bad things happened"),
}

func bad() bool {
    return false
}

func returnsError() error {
    var p *MyError = nil
    if bad() {
        p = &ErrBad
    }
    return p
}

func main() {
    err := returnsError()
    if err != nil {
        fmt.Printf("error occur: %+v\n", err)
        return
    }
    fmt.Println("ok")
}
```

在这个例子中，我们的关注点集中在 returnsError 这个函数上面。这个函数定义了一个*MyError类型的变量 p，初值为 nil。如果函数 bad 返回 false，returnsError 函数就会直接将 p（此时 p = nil）作为返回值返回给调用者，之后调用者会将 returnsError 函数的返回值（error 接口类型）与 nil 进行比较，并根据比较结果做出最终处理。

如果你是一个初学者，我猜你的的思路大概是这样的：p 为 nil，returnsError 返回 p，那么 main 函数中的 err 就等于 nil，于是程序输出 **ok** 后退出。

但真实的运行结果是什么样的呢？我们来看一下：

复制代码

```
error occur: <nil>
```

我们看到，示例程序并未如我们前面预期的那样输出 ok。程序显然是进入了错误处理分支，输出了 err 的值。那这里就有一个问题了：明明 returnsError 函数返回的 p 值为 nil，为什么却满足了`if err != nil`的条件进入错误处理分支呢？

要想弄清楚这个问题，我们需要进一步了解接口类型变量的内部表示。

## 接口类型变量的内部表示

接口类型“动静兼备”的特性也决定了它的变量的内部表示绝不像一个静态类型变量（如 int、float64）那样简单，我们可以在`$GOROOT/src/runtime/runtime2.go`中找到接口类型变量在运行时的表示：

复制代码

```
// $GOROOT/src/runtime/runtime2.go
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

我们看到，在运行时层面，接口类型变量有两种内部表示：iface和eface，这两种表示分别用于不同的接口类型变量：

> eface 用于表示没有方法的空接口（**e**mpty inter**face**）类型变量，也就是 interface{}类型的变量；
>
> iface 用于表示其余拥有方法的接口 **i**nter**face** 类型变量。

这两个结构的共同点是它们都有两个指针字段，并且第二个指针字段的功能相同，都是指向当前赋值给该接口类型变量的动态类型变量的值。

那它们的不同点在哪呢？就在于 eface 表示的空接口类型并没有方法列表，因此它的第一个指针字段指向一个_type类型结构，这个结构为该接口类型变量的动态类型的信息，它的定义是这样的：

复制代码

```
// $GOROOT/src/runtime/type.go

type _type struct {
    size       uintptr
    ptrdata    uintptr // size of memory prefix holding all pointers
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    // function for comparing objects of this type
    // (ptr to object A, ptr to object B) -> ==?
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    // gcdata stores the GC type data for the garbage collector.
    // If the KindGCProg bit is set in kind, gcdata is a GC program.
    // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
    gcdata    *byte
    str       nameOff
    ptrToThis typeOff
}
```

而 iface 除了要存储动态类型信息之外，还要存储接口本身的信息（接口的类型信息、方法列表信息等）以及动态类型所实现的方法的信息，因此 iface 的第一个字段指向一个itab类型结构。itab 结构的定义如下：

复制代码

```
// $GOROOT/src/runtime/runtime2.go
type itab struct {
    inter *interfacetype
    _type *_type
    hash  uint32 // copy of _type.hash. Used for type switches.
    _     [4]byte
    fun   [1]uintptr // variable sized. fun[0]==0 means _type does not impleme
}
```

这里我们也可以看到，itab 结构中的第一个字段inter指向的 interfacetype 结构，存储着这个接口类型自身的信息。你看一下下面这段代码表示的 interfacetype 类型定义， 这个 interfacetype 结构由类型信息（typ）、包路径名（pkgpath）和接口方法集合切片（mhdr）组成。

复制代码

```
// $GOROOT/src/runtime/type.go
type interfacetype struct {
    typ     _type
    pkgpath name
    mhdr    []imethod
}
```

itab 结构中的字段_type则存储着这个接口类型变量的动态类型的信息，字段fun则是动态类型已实现的接口方法的调用地址数组。

下面我们再结合例子用图片来直观展现 eface 和 iface 的结构。首先我们看一个用 eface 表示的空接口类型变量的例子：

复制代码

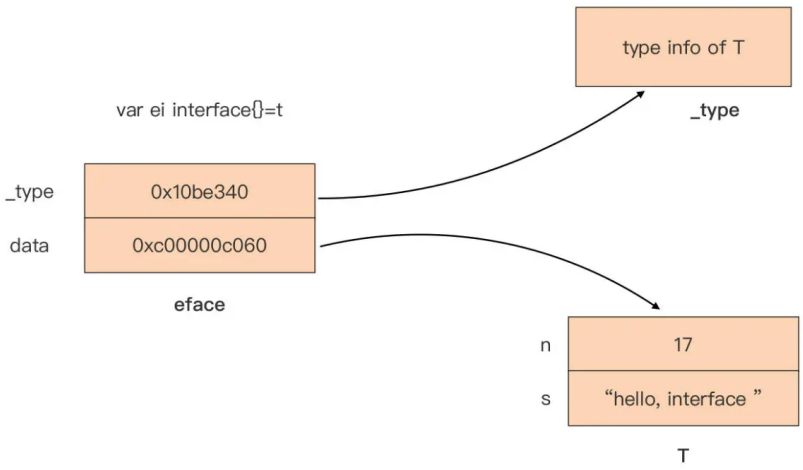
```
type T struct {
    n int
    s string
}

func main() {
    var t = T {
        n: 17,
        s: "hello, interface",
    }

```

```
12      var ei interface{} = t // Go运行时使用eface结构表示ei
13  }
```

这个例子中的空接口类型变量 ei 在 Go 运行时的表示是这样的：




极客时间

我们看到空接口类型的表示较为简单，图中上半部分 _type 字段指向它的动态类型 T 的类型信息，下半部分的 data 则是指向一个 T 类型的实例值。

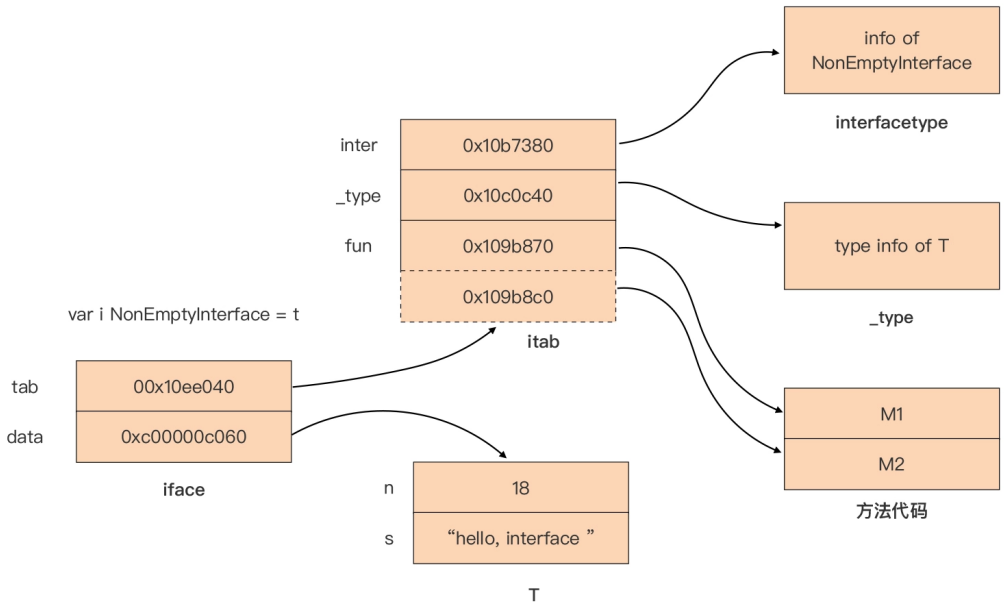
我们再来看一个更复杂的用 iface 表示非空接口类型变量的例子：

复制代码

```
1  type T struct {
2      n int
3      s string
4  }
5
6  func (T) M1() {}
7  func (T) M2() {}
8
9  type NonEmptyInterface interface {
10     M1()
11     M2()
12 }
13
14 func main() {
15     var t = T{
16         n: 18,
```

```
17          s: "hello, interface",
18      }
19      var i NonEmptyInterface = t
20  }
```

和 eface 比起来，iface 的表示稍微复杂些。我也画了一幅表示上面 NonEmptyInterface 接口类型变量在 Go 运行时表示的示意图：




极客时间

由上面的这两幅图，我们可以看出，每个接口类型变量在运行时的表示都是由两部分组成的，针对不同接口类型我们可以简化记作：`eface(_type, data)`和`iface(tab, data)`。

而且，虽然 eface 和 iface 的第一个字段有所差别，但 tab 和 _type 可以统一看作是动态类型的类型信息。Go 语言中每种类型都会有唯一的 _type 信息，无论是内置原生类型，还是自定义类型都有。Go 运行时会为程序内的全部类型建立只读的共享 _type 信息表，因此拥有相同动态类型的同类接口类型变量的 _type/tab 信息是相同的。

而接口类型变量的 data 部分则是指向一个动态分配的内存空间，这个内存空间存储的是赋值给接口类型变量的动态类型变量的值。未显式初始化的接口类型变量的值为`nil`，也就是这个变量的 _type/tab 和 data 都为 nil。

也就是说，我们判断两个接口类型变量是否相同，只需要判断 _type/tab 是否相同，以及 data 指针指向的内存空间所存储的数据值是否相同就可以了。这里要注意不是 data 指针的值相同噢。

不过，通过肉眼去辨别接口类型变量是否相等总是困难一些，我们可以引入一些 **helper 函数**。借助这些函数，我们可以清晰地输出接口类型变量的内部表示，这样就可以一目了然地看出两个变量是否相等了。

由于 eface 和 iface 是 runtime 包中的非导出结构体定义，我们不能直接在包外使用，所以也就无法直接访问到两个结构体中的数据。不过，Go 语言提供了 println 预定义函数，可以用来输出 eface 或 iface 的两个指针字段的值。

在编译阶段，编译器会根据要输出的参数的类型将 println 替换为特定的函数，这些函数都定义在`$GOROOT/src/runtime/print.go`文件中，而针对 eface 和 iface 类型的打印函数实现如下：

复制代码

```
// $GOROOT/src/runtime/print.go
func printeface(e eface) {
    print("(", e._type, ",", e.data, ")")
}

func printiface(i iface) {
    print("(", i.tab, ",", i.data, ")")
}
```

我们看到，printeface 和 printiface 会输出各自的两个指针字段的值。下面我们就来使用 println 函数输出各类接口类型变量的内部表示信息，并结合输出结果，解析接口类型变量的等值比较操作。

## 第一种：nil 接口变量

我们前面提过，未赋初值的接口类型变量的值为 nil，这类变量也就是 nil 接口变量，我们来看这类变量的内部表示输出的例子：

复制代码

```
func printNilInterface() {
```

```
    // nil接口变量
    var i interface{} // 空接口类型
    var err error     // 非空接口类型
    println(i)
    println(err)
    println("i = nil:", i == nil)
    println("err = nil:", err == nil)
    println("i = err:", i == err)
}
```

运行这个函数，输出结果是这样的：

复制代码

```
(0x0,0x0)
(0x0,0x0)
i = nil: true
err = nil: true
i = err: true
```

我们看到，无论是空接口类型还是非空接口类型变量，一旦变量值为 nil，那么它们内部表示均为(0x0,0x0)，也就是类型信息、数据值信息均为空。因此上面的变量 i 和 err 等值判断为 true。

## 第二种：空接口类型变量

下面是空接口类型变量的内部表示输出的例子：

复制代码

```
func printEmptyInterface() {
    var eif1 interface{} // 空接口类型
    var eif2 interface{} // 空接口类型
    var n, m int = 17, 18

    eif1 = n
    eif2 = m

    println("eif1:", eif1)
    println("eif2:", eif2)
    println("eif1 = eif2:", eif1 == eif2) // false

    eif2 = 17
    println("eif1:", eif1)
    println("eif2:", eif2)
```

```
16        println("eif1 = eif2:", eif1 == eif2) // true
17
18        eif2 = int64(17)
19        println("eif1:", eif1)
20        println("eif2:", eif2)
21        println("eif1 = eif2:", eif1 == eif2) // false
22    }
```

这个例子的运行输出结果是这样的：

复制代码

```
1 eif1: (0x10ac580,0xc00007ef48)
2 eif2: (0x10ac580,0xc00007ef40)
3 eif1 = eif2: false
4 eif1: (0x10ac580,0xc00007ef48)
5 eif2: (0x10ac580,0x10eb3d0)
6 eif1 = eif2: true
7 eif1: (0x10ac580,0xc00007ef48)
8 eif2: (0x10ac640,0x10eb3d8)
9 eif1 = eif2: false
```

我们按顺序分析一下这个输出结果。

首先，代码执行到第 11 行时，eif1 与 eif2 已经分别被赋值整型值 17 与 18，这样 eif1 和 eif2 的动态类型的类型信息是相同的（都是 0x10ac580），但 data 指针指向的内存块中存储的值不同，一个是 17，一个是 18，于是 eif1 不等于 eif2。

接着，代码执行到第 16 行的时候，eif2 已经被重新赋值为 17，这样 eif1 和 eif2 不仅存储的动态类型的类型信息是相同的（都是 0x10ac580），data 指针指向的内存块中存储值也相同了，都是 17，于是 eif1 等于 eif2。

然后，代码执行到第 21 行时，eif2 已经被重新赋值了 int64 类型的数值 17。这样，eif1 和 eif2 存储的动态类型的类型信息就变成不同的了，一个是 int，一个是 int64，即便 data 指针指向的内存块中存储值是相同的，最终 eif1 与 eif2 也是不相等的。

从输出结果中我们可以总结一下：**对于空接口类型变量，只有 _type 和 data 所指数据内容一致的情况下，两个空接口类型变量之间才能划等号。**另外，Go 在创建 eface 时一般会为 data 重新分配新内存空间，将动态类型变量的值复制到这块内存空间，并将 data 指针指向这块内存空间。因此我们多数情况下看到的 data 指针值都是不同的。

## 第三种：非空接口类型变量

这里，我们也直接来看一个非空接口类型变量的内部表示输出的例子：

复制代码

```
type T int

func (t T) Error() string {
    return "bad error"
}

func printNonEmptyInterface() {
    var err1 error // 非空接口类型
    var err2 error // 非空接口类型
    err1 = (*T)(nil)
    println("err1:", err1)
    println("err1 = nil:", err1 == nil)

    err1 = T(5)
    err2 = T(6)
    println("err1:", err1)
    println("err2:", err2)
    println("err1 = err2:", err1 == err2)

    err2 = fmt.Errorf("%d\n", 5)
    println("err1:", err1)
    println("err2:", err2)
    println("err1 = err2:", err1 == err2)
}
```

这个例子的运行输出结果如下：

复制代码

```
err1: (0x10ed120,0x0)
err1 = nil: false
err1: (0x10ed1a0,0x10eb310)
err2: (0x10ed1a0,0x10eb318)
err1 = err2: false
err1: (0x10ed1a0,0x10eb310)
err2: (0x10ed0c0,0xc000010050)
err1 = err2: false
```

我们看到上面示例中每一轮通过 println 输出的 err1 和 err2 的 tab 和 data 值，要么 data 值不同，要么 tab 与 data 值都不同。

和空接口类型变量一样，只有 tab 和 data 指的数据内容一致的情况下，两个非空接口类型变量之间才能划等号。这里我们要注意 err1 下面的赋值情况：

复制代码

```
err1 = (*T)(nil)
```

针对这种赋值，println 输出的 err1 是（0x10ed120, 0x0），也就是非空接口类型变量的类型信息并不为空，数据指针为空，因此它与 nil（0x0,0x0）之间不能划等号。

现在我们再回到我们开头的那个问题，你是不是已经豁然开朗了呢？开头的问题中，从 returnsError 返回的 error 接口类型变量 err 的数据指针虽然为空，但它的类型信息（iface.tab）并不为空，而是 *MyError 对应的类型信息，这样 err 与 nil（0x0,0x0）相比自然不相等，这就是我们开头那个问题的答案解析，现在你明白了吗？

## 第四种：空接口类型变量与非空接口类型变量的等值比较

下面是非空接口类型变量和空接口类型变量之间进行比较的例子：

复制代码

```
func printEmptyInterfaceAndNonEmptyInterface() {
  var eif interface{} = T(5)
  var err error = T(5)
  println("eif:", eif)
  println("err:", err)
  println("eif = err:", eif == err)

  err = T(6)
  println("eif:", eif)
  println("err:", err)
  println("eif = err:", eif == err)
}
```

这个示例的输出结果如下：

复制代码

```
eif: (0x10b3b00,0x10eb4d0)
err: (0x10ed380,0x10eb4d8)
eif = err: true
eif: (0x10b3b00,0x10eb4d0)
```

```
5 err: (0x10ed380,0x10eb4e0)
6 eif = err: false
```

你可以看到，空接口类型变量和非空接口类型变量内部表示的结构有所不同（第一个字段：_type vs. tab)，两者似乎一定不能相等。但 Go 在进行等值比较时，类型比较使用的是 eface 的 _type 和 iface 的 tab._type，因此就像我们在这个例子中看到的那样，当 eif 和 err 都被赋值为T(5)时，两者之间是划等号的。

好了，到这里，我们已经学完了各类接口类型变量在运行时层的表示。我们可以通过 println 可以查看这个表示信息，从中我们也知道了接口变量只有在类型信息与值信息都一致的情况下才能划等号。

## 输出接口类型变量内部表示的详细信息

不过，println 输出的接口类型变量的内部表示信息，在一般情况下都是足够的，但有些时候又显得过于简略，比如在上面最后一个例子中，如果仅凭`eif: (0x10b3b00,0x10eb4d0)`和`err: (0x10ed380,0x10eb4d8)`的输出，我们是无法想到两个变量是相等的。

那这时如果我们能输出接口类型变量内部表示的详细信息（比如：tab._type），那势必可以取得事半功倍的效果。接下来我们就看看这要怎么做。

前面提到过，eface 和 iface 以及组成它们的 itab 和 _type 都是 runtime 包下的非导出结构体，我们无法在外部直接引用它们。但我们发现，组成 eface、iface 的类型都是基本数据类型，我们完全可以通过**“复制代码”**的方式将它们拿到 runtime 包外面来。

不过，这里要注意，由于 runtime 中的 eface、iface，或者它们的组成可能会随着 Go 版本的变化发生变化，因此这个方法不具备跨版本兼容性。也就是说，基于 Go 1.17 版本复制的代码，可能仅适用于使用 Go 1.17 版本编译。这里我们就以 Go 1.17 版本为例看看：

复制代码

```
1 // dumpinterface.go
2 type eface struct {
3     _type *_type
4     data  unsafe.Pointer
5 }
6
```

```
type tflag uint8
type nameOff int32
type typeOff int32

type _type struct {
    size       uintptr
    ptrdata    uintptr // size of memory prefix holding all pointers
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    // function for comparing objects of this type
    // (ptr to object A, ptr to object B) -> ==?
    equal func(unsafe.Pointer, unsafe.Pointer) bool
    // gcdata stores the GC type data for the garbage collector.
    // If the KindGCProg bit is set in kind, gcdata is a GC program.
    // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
    gcdata    *byte
    str       nameOff
    ptrToThis typeOff
}

type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type itab struct {
    inter *interfacetype
    _type *_type
    hash  uint32 // copy of _type.hash. Used for type switches.
    _     [4]byte
    fun   [1]uintptr // variable sized. fun[0]==0 means _type does not impleme
}

... ...

const ptrSize = unsafe.Sizeof(uintptr(0))

func dumpEface(i interface{}) {
    ptrToEface := (*eface)(unsafe.Pointer(&i))
    fmt.Printf("eface: %+v\n", *ptrToEface)

    if ptrToEface._type != nil {
        // dump _type info
        fmt.Printf("\t _type: %+v\n", *(ptrToEface._type))
    }

    if ptrToEface.data != nil {
        // dump data
        switch i.(type) {
```

```
        case int:
            dumpInt(ptrToEface.data)
        case float64:
            dumpFloat64(ptrToEface.data)
        case T:
            dumpT(ptrToEface.data)

        // other cases ... ...
        default:
            fmt.Printf("\t unsupported data type\n")
        }
    }
    fmt.Printf("\n")
}

func dumpItabOfIface(ptrToIface unsafe.Pointer) {
    p := (*iface)(ptrToIface)
    fmt.Printf("iface: %+v\n", *p)

    if p.tab != nil {
        // dump itab
        fmt.Printf("\t itab: %+v\n", *(p.tab))
        // dump inter in itab
        fmt.Printf("\t\t inter: %+v\n", *(p.tab.inter))

        // dump _type in itab
        fmt.Printf("\t\t _type: %+v\n", *(p.tab._type))

        // dump fun in tab
        funPtr := unsafe.Pointer(&(p.tab.fun))
        fmt.Printf("\t\t fun: [")
        for i := 0; i < len((*(p.tab.inter)).mhdr); i++ {
            tp := (*uintptr)(unsafe.Pointer(uintptr(funPtr) + uintptr(i)*ptrSi
            fmt.Printf("0x%x(%d),", *tp, *tp)
        }
        fmt.Printf("]\n")
    }
}

func dumpDataOfIface(i interface{}) {
    // this is a trick as the data part of eface and iface are same
    ptrToEface := (*eface)(unsafe.Pointer(&i))

    if ptrToEface.data != nil {
        // dump data
        switch i.(type) {
        case int:
            dumpInt(ptrToEface.data)
        case float64:
            dumpFloat64(ptrToEface.data)
        case T:
            dumpT(ptrToEface.data)
```

```
111
112         // other cases ... ...
113
114         default:
115             fmt.Printf("\t unsupported data type\n")
116         }
117     }
118     fmt.Printf("\n")
119 }
120
121 func dumpT(dataOfIface unsafe.Pointer) {
122     var p *T = (*T)(dataOfIface)
123     fmt.Printf("\t data: %+v\n", *p)
124 }
125 ... ...
126
```

这里我挑选了关键部分，省略了部分代码。上面这个 dumpinterface.go 中提供了三个主要函数:

dumpEface: 用于输出空接口类型变量的内部表示信息；

dumpItabOfIface: 用于输出非空接口类型变量的 tab 字段信息；

dumpDataOfIface: 用于输出非空接口类型变量的 data 字段信息；

我们利用这三个函数来输出一下前面 printEmptyInterfaceAndNonEmptyInterface 函数中的接口类型变量的信息：

复制代码

```
1  package main
2
3  import "unsafe"
4
5  type T int
6
7  func (t T) Error() string {
8      return "bad error"
9  }
10
11 func main() {
12     var eif interface{} = T(5)
13     var err error = T(5)
14     println("eif:", eif)
15     println("err:", err)
16     println("eif = err:", eif == err)
```

```
17
18      dumpEface(eif)
19      dumpItabOfIface(unsafe.Pointer(&err))
20      dumpDataOfIface(err)
21  }
```

运行这个示例代码，我们得到了这个输出结果：

复制代码

```
1  eif: (0x10b38c0,0x10e9b30)
2  err: (0x10eb690,0x10e9b30)
3  eif = err: true
4  eface: {_type:0x10b38c0 data:0x10e9b30}
5     _type: {size:8 ptrdata:0 hash:1156555957 tflag:15 align:8 fieldAlign:8 kind
6     data: bad error
7
8  iface: {tab:0x10eb690 data:0x10e9b30}
9     itab: {inter:0x10b5e20 _type:0x10b38c0 hash:1156555957 _:[0 0 0 0] fun:[174
10      inter: {typ:{size:16 ptrdata:16 hash:235953867 tflag:7 align:8 fieldAlign
11      _type: {size:8 ptrdata:0 hash:1156555957 tflag:15 align:8 fieldAlign:8 ki
12      fun: [0x10a5780(17454976),]
13    data: bad error
```

从输出结果中，我们看到 eif 的 _type（0x10b38c0）与 err 的 tab._type（0x10b38c0）是一致的，data 指针所指内容（“bad error”）也是一致的，因此`eif == err`表达式的结果为 true。

再次强调一遍，上面这个实现可能仅在 Go 1.17 版本上测试通过，并且在输出 iface 或 eface 的 data 部分内容时只列出了 int、float64 和 T 类型的数据读取实现，没有列出全部类型的实现，你可以根据自己的需要实现其余数据类型。dumpinterface.go 的完整代码你可以在这里找到。

我们现在已经知道了，接口类型有着复杂的内部结构，所以我们将一个类型变量值赋值给一个接口类型变量值的过程肯定不会像`var i int = 5`那么简单，那么接口类型变量赋值的过程是怎样的呢？其实接口类型变量赋值是一个“装箱”的过程。

## 接口类型的装箱（boxing）原理

**装箱（boxing）**是编程语言领域的一个基础概念，一般是指把一个值类型转换成引用类型，比如在支持装箱概念的 Java 语言中，将一个 int 变量转换成 Integer 对象就是一个装箱操作。

在 Go 语言中，将任意类型赋值给一个接口类型变量也是**装箱**操作。有了前面对接口类型变量内部表示的学习，我们知道**接口类型的装箱实际就是创建一个 eface 或 iface 的过程**。接下来我们就来简要描述一下这个过程，也就是接口类型的装箱原理。

我们基于下面这个例子中的接口装箱操作来说明：

复制代码

```
// interface_internal.go

  type T struct {
      n int
      s string
  }

  func (T) M1() {}
  func (T) M2() {}

  type NonEmptyInterface interface {
      M1()
      M2()
  }

  func main() {
      var t = T{
          n: 17,
          s: "hello, interface",
      }
      var ei interface{}
      ei = t

      var i NonEmptyInterface
      i = t
      fmt.Println(ei)
      fmt.Println(i)
  }
```

这个例子中，对 ei 和 i 两个接口类型变量的赋值都会触发装箱操作，要想知道 Go 在背后做了些什么，我们需要“下沉”一层，也就是要输出上面 Go 代码对应的汇编代码：

复制代码

```
$go tool compile -S interface_internal.go > interface_internal.s
```

对应ei = t一行的汇编如下：

复制代码

```
    0x0026 00038 (interface_internal.go:24) MOVQ    $17, ""..autotmp_15+104(SP
    0x002f 00047 (interface_internal.go:24) LEAQ    go.string."hello, interfac
    0x0036 00054 (interface_internal.go:24) MOVQ    CX, ""..autotmp_15+112(SP)
    0x003b 00059 (interface_internal.go:24) MOVQ    $16, ""..autotmp_15+120(SP
    0x0044 00068 (interface_internal.go:24) LEAQ    type."".T(SB), AX
    0x004b 00075 (interface_internal.go:24) LEAQ    ""..autotmp_15+104(SP), BX
    0x0050 00080 (interface_internal.go:24) PCDATA  $1, $0
    0x0050 00080 (interface_internal.go:24) CALL    runtime.convT2E(SB)
```

对应 i = t 一行的汇编如下：

复制代码

```
    0x005f 00095 (interface_internal.go:27) MOVQ    $17, ""..autotmp_15+104(SP
    0x0068 00104 (interface_internal.go:27) LEAQ    go.string."hello, interfac
    0x006f 00111 (interface_internal.go:27) MOVQ    CX, ""..autotmp_15+112(SP)
    0x0074 00116 (interface_internal.go:27) MOVQ    $16, ""..autotmp_15+120(SP
    0x007d 00125 (interface_internal.go:27) LEAQ    go.itab."".T,"".NonEmptyIn
    0x0084 00132 (interface_internal.go:27) LEAQ    ""..autotmp_15+104(SP), BX
    0x0089 00137 (interface_internal.go:27) PCDATA  $1, $1
    0x0089 00137 (interface_internal.go:27) CALL    runtime.convT2I(SB)
```

在将动态类型变量赋值给接口类型变量语句对应的汇编代码中，我们看到了convT2E和convT2I两个 runtime 包的函数。这两个函数的实现位于$GOROOT/src/runtime/iface.go中：

复制代码

```
// $GOROOT/src/runtime/iface.go

func convT2E(t *_type, elem unsafe.Pointer) (e eface) {
    if raceenabled {
        raceReadObjectPC(t, elem, getcallerpc(), funcPC(convT2E))
    }
    if msanenabled {
        msanread(elem, t.size)
```

```
    }
    x := mallocgc(t.size, t, true)
    typedmemmove(t, x, elem)
    e._type = t
    e.data = x
    return
}

func convT2I(tab *itab, elem unsafe.Pointer) (i iface) {
    t := tab._type
    if raceenabled {
        raceReadObjectPC(t, elem, getcallerpc(), funcPC(convT2I))
    }
    if msanenabled {
        msanread(elem, t.size)
    }
    x := mallocgc(t.size, t, true)
    typedmemmove(t, x, elem)
    i.tab = tab
    i.data = x
    return
}
```

convT2E 用于将任意类型转换为一个 eface，convT2I 用于将任意类型转换为一个 iface。两个函数的实现逻辑相似，主要思路就是根据传入的类型信息（convT2E 的 _type 和 convT2I 的 tab._type）分配一块内存空间，并将 elem 指向的数据拷贝到这块内存空间中，最后传入的类型信息作为返回值结构中的类型信息，返回值结构中的数据指针（data）指向新分配的那块内存空间。

由此我们也可以看出，经过装箱后，箱内的数据，也就是存放在新分配的内存空间中的数据与原变量便无瓜葛了，比如下面这个例子：

复制代码

```
func main() {
  var n int = 61
  var ei interface{} = n
  n = 62  // n的值已经改变
  fmt.Println("data in box:", ei) // 输出仍是61
}
```

那么 convT2E 和 convT2I 函数的类型信息是从何而来的呢？

其实这些都依赖 Go 编译器的工作。编译器知道每个要转换为接口类型变量（toType）和动态类型变量的类型（fromType），它会根据这一对类型选择适当的 convT2X 函数，并在生成代码时使用选出的 convT2X 函数参与装箱操作。

不过，装箱是一个有性能损耗的操作，因此 Go 也在不断对装箱操作进行优化，包括对常见类型如整型、字符串、切片等提供系列快速转换函数：

复制代码

```
// $GOROOT/src/runtime/iface.go
func convT16(val any) unsafe.Pointer     // val must be uint16-like
func convT32(val any) unsafe.Pointer     // val must be uint32-like
func convT64(val any) unsafe.Pointer     // val must be uint64-like
func convTstring(val any) unsafe.Pointer // val must be a string
func convTslice(val any) unsafe.Pointer  // val must be a slice
```

这些函数去除了 typedmemmove 操作，增加了零值快速返回等特性。

同时 Go 建立了 staticuint64s 区域，对 255 以内的小整数值进行装箱操作时不再分配新内存，而是利用 staticuint64s 区域的内存空间，下面是 staticuint64s 的定义：

复制代码

```
// $GOROOT/src/runtime/iface.go
// staticuint64s is used to avoid allocating in convTx for small integer value
var staticuint64s = [...]uint64{
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
  ... ...
}
```

## 小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

接口类型作为参与构建 Go 应用骨架的重要参与者，在 Go 语言中有着很高的地位。它这个地位的取得离不开它拥有的“动静兼备”的语法特性。Go 接口的动态特性让 Go 拥有与动态语言相近的灵活性，而静态特性又在编译阶段保证了这种灵活性的安全。

要更好地理解 Go 接口的这两种特性，我们需要深入到 Go 接口在运行时的表示层面上去。接口类型变量在运行时表示为 eface 和 iface，eface 用于表示空接口类型变量，iface 用于表示非空接口类型变量。只有两个接口类型变量的类型信息（eface._type/iface.tab._type）相同，且数据指针（eface.data/iface.data）所指数据相同时，两个接口类型变量才是相等的。

我们可以通过 println 输出接口类型变量的两部分指针变量的值。而且，通过拷贝 runtime 包 eface 和 iface 相关类型源码，我们还可以自定义输出 eface/iface 详尽信息的函数，不过要注意的是，由于 runtime 层代码的演进，这个函数可能不具备在 Go 版本间的移植性。

最后，接口类型变量的赋值本质上是一种装箱操作，装箱操作是由 Go 编译器和运行时共同完成的，有一定的性能开销，对于性能敏感的系统来说，我们应该尽量避免或减少这类装箱操作。

## 思考题

像 nil error 值 != nil 那个例子中的“坑”你在日常编码时有遇到过吗？可以和我们分享一下吗？另外，我们这节课中的这个例子如何修改，才能让它按我们最初的预期结果输出呢？

欢迎在留言区分享你的经验和想法。也欢迎你把这节课分享给更多对 Go 接口感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 28 | 接口：接口即契约

下一篇

下一篇

更多课程推荐

## 精选留言 (8)

写留言

**return**
2021-12-29

老师讲的太好， 这一篇 知识密度相当大啊，
就这一篇就值专栏的价格了。
感谢老师如此用心的输出。

2

**Geralt of Rivia**
2021-12-29

修改方法:
1. 把returnsError()里面p的类型改为error
2. 删除p，直接return &ErrBad或者nil

共 1 条评论 >

2

**i6u**
2021-12-29

返回 *MyError 而不是 error

共 1 条评论 >

1

**在下宝龙、**

2021-12-29

老师您好，在 eif2 = 17 这个操作后，输出后的data ,0xc00007ef48 和0x10eb3d0 不相等呀，为甚么说他们是一样的
eif1: (0x10ac580,0xc00007ef48)
eif2: (0x10ac580,0x10eb3d0)

共 2 条评论 >

1

**bearlu**

2021-12-29

这次课很干。需要再学一遍

1

**aoe**

2022-01-03

原来装箱是这样：将任意类型赋值给一个接口类型变量就是装箱操作。
接口类型的装箱实际就是创建一个 eface 或 iface 的过程

**木木**

2022-01-02

这节学到很多！

**lesserror**

2021-12-31

大白老师的这一节干货很多，读的意犹未尽。有几个疑惑点，麻烦老师解忧。

1. 文中类似："_type" 这种命名，前面加下划线，这种有什么含义呢？

2. 文中关于打印两类接口内部详细信息的代码中，运用了大量的 * 还有 & 再加上 unsaf...

展开